

## AI Assisted Coding

### Lab\_Assignment\_2.3

**2303A51383**

**Batch-06**

#### PROMPT:

##### Task 1: Word Frequency from Text File

###### ❖ Scenario:

You are analyzing log files for keyword frequency.

###### ❖ Task:

Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code.

###### ❖ Expected Output:

➤ Working code

➤ Explanation

➤ Screenshot

#### Code:

```
# First, let's create a dummy text file for demonstration purposes.
```

```
file_content = """This is a sample text file.
```

```
This file will be used to count word frequencies.
```

```
Sample text, sample words, sample frequency."""
```

```
with open('sample_log.txt', 'w') as f:
```

```

f.write(file_content)

print("Created 'sample_log.txt' with sample content.")


import re
from collections import Counter
def count_word_frequency(filepath): """
    Reads a text file and counts the frequency of each word.

    Args: filepath (str): The path to the text
        file.

    Returns:
        collections.Counter: A Counter object where keys are words and values are
        their frequencies.

    """ words = [] with
    open(filepath, 'r') as f:
        for line in f:
            # Convert to lowercase and remove punctuation, then split into words
            # re.findall('\b\w+\b', ...) finds all word characters
            words.extend(re.findall(r'\b\w+\b', line.lower()))

    return Counter(words)


# Specify the path to the text file

file_path = 'sample_log.txt' #

Count word frequencies

```

```
word_counts =  
  
count_word_frequency(file_path)  
  
h)  
  
# Display the top 10 most common words  
print("\nWord Frequencies (Top 10):") for word,  
count in word_counts.most_common(10):  
print(f"{word}: {count}")
```

## OUTPUT:

Word Frequencies (Top 10):

'sample': 4

'this': 2

'text': 2

'file': 2

'is': 1

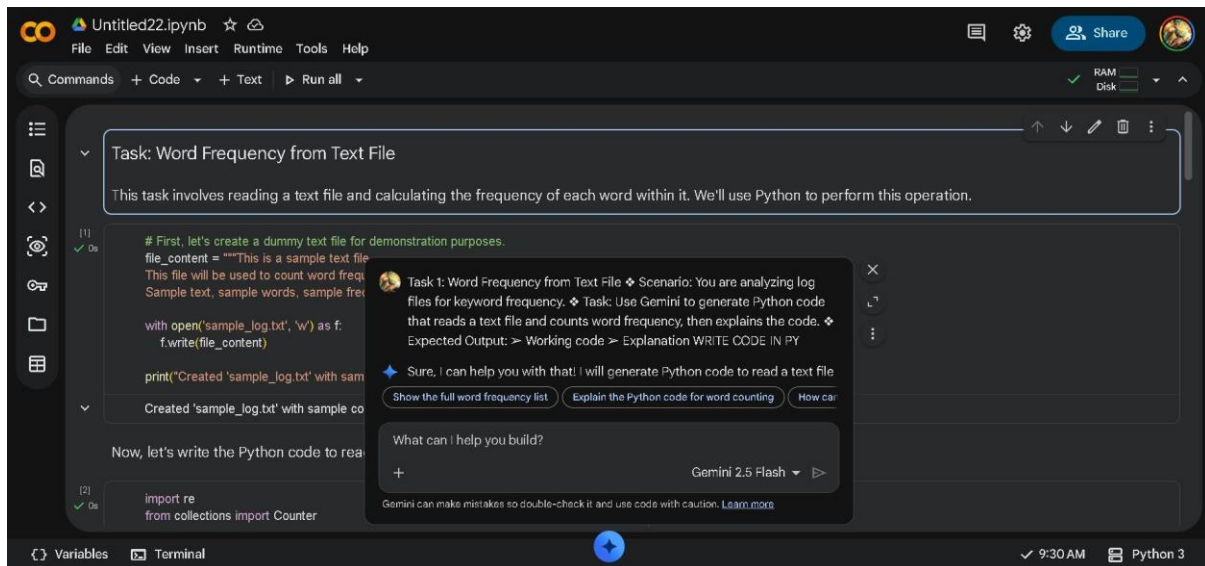
'a': 1

'will': 1

'be': 1

'used': 1

'to': 1



### ### Explanation of the Code:

#### 1. **\*\*Import necessary libraries:\*\***

- \* ``re``: The regular expression module is used for robust word extraction (splitting text into words while handling punctuation). ``re.findall(r'\b\w+\b', ...)`` finds all sequences of word characters. ``\b`` matches a word boundary, and ``\w+`` matches one or more alphanumeric characters (letters, numbers, and underscores).

- \* ``collections.Counter``: This is a specialized dictionary subclass for counting hashable objects. It's very efficient for frequency counting tasks.

#### 2. **\*\*``count_word_frequency(filepath)`` function:\*\*** \* Takes ``filepath`` as an argument.

- \* Initializes an empty list ``words`` to store all extracted words.
- \* Opens the specified file in read mode (``'r'``). The ``with`` statement ensures the file is properly closed even if errors occur.
- \* Iterates through each ``line`` in the file.
- \* For each line:

- \* `line.lower()`: Converts the line to lowercase to ensure that words like "The" and "the" are counted as the same word.
- \* `re.findall(r'\b\w+\b', ...)`: Uses a regular expression to find all words. This effectively removes punctuation and splits the text into individual words.
- \* `words.extend(...)`: Adds the list of words found in the current line to the overall `words` list.
- \* Finally, `return Counter(words)` creates and returns a `Counter` object from the collected list of words, which automatically calculates their frequencies.

### 3. **\*\*Execution Block:\*\***

- \* `file_path = 'sample_log.txt'`: Defines the name of the file to be analyzed.
- \* `word_counts = count_word_frequency(file_path)`: Calls the function to get the word frequencies.
- \* The code then prints the top 10 most common words and their counts using `word_counts.most_common(10)`.

## Task 2: File Operations Using Cursor AI

### ❖ Scenario:

You are automating basic file operations.

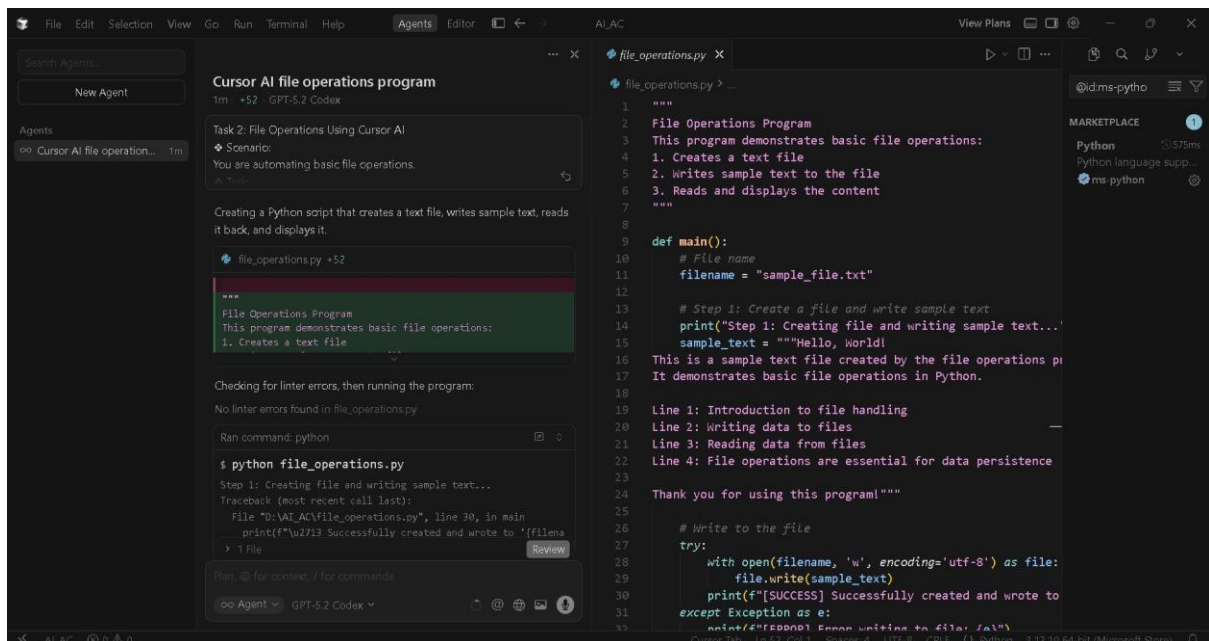
### ❖ Task:

Use Cursor AI to generate a program that:

- Creates a text file

- Writes sample text
- Reads and displays the content
- ❖ Expected Output:
- Functional code ➤

## Cursor AI screenshots



## CODE:

```
"""
```

### File Operations Program

This program demonstrates basic file operations:

1. Creates a text file
2. Writes sample text to the file
3. Reads and displays the content """

```
def main(): #
```

*File name*

```
filename = "sample_file.txt"
```

```
# Step 1: Create a file and write sample text
```

```
print("Step 1: Creating file and writing sample text...")
```

```
sample_text = """Hello, World!
```

This is a sample text file created by the file operations program.

It demonstrates basic file operations in Python.

Line 1: Introduction to file handling

Line 2: Writing data to files

Line 3: Reading data from files

Line 4: File operations are essential for data persistence

```
Thank you for using this program!"""
```

```
# Write to the file
```

```
try:
```

```
    with open(filename, 'w', encoding='utf-8') as file:
```

```
        file.write(sample_text) print(f"[SUCCESS] Successfully created  
and wrote to '{filename}')) except Exception as e:
```

```
    print(f"[ERROR] Error writing to file: {e}")
```

```
    return
```

```
# Step 2: Read and display the content
```

```
print("\nStep 2: Reading and displaying file content...")
```

```
try:
```

```
    with open(filename, 'r', encoding='utf-8') as file:
```

```
        content = file.read()
```

```

        print(f"\n--- Content of '{filename}' ---")

        print(content) print("--- End of file content ---")

    print(f"\n[SUCCESS] Successfully read '{filename}'")

except FileNotFoundError: print(f"[ERROR] File

'{filename}' not found") except Exception as e:

print(f"[ERROR] Error reading file: {e}")

if __name__ == "__main__":

    main()

```

## OUTPUT:

```

● Step 1: Creating file and writing sample text...
[SUCCESS] Successfully created and wrote to 'sample_file.txt'

Step 2: Reading and displaying file content...

--- Content of 'sample_file.txt' ---
Hello, World!
This is a sample text file created by the file operations program.
It demonstrates basic file operations in Python.

Line 1: Introduction to file handling
Line 2: Writing data to files
Line 3: Reading data from files
Line 4: File operations are essential for data persistence

Thank you for using this program!
--- End of file content ---

[SUCCESS] Successfully read 'sample_file.txt'

```

## EXPLANATION:

Code Explanation: file\_operations.py

Python script that demonstrates basic file operations. Here's what it does:



## Overall Structure

The program performs two main tasks:

1. Create and write to a text file
2. Read and display the file's contents

## Breakdown by Section

Lines 1-7: Documentation

- Module-level docstring describing the program's purpose Lines 9-33: File

## Writing Section

Lines 11-24:

- filename: Target file name ("sample\_file.txt")
- sample\_text: Multi-line string containing sample content Lines 27-33: Write

Operation *with* open(filename, 'w', encoding='utf-8') as file:

```
file.write(sample_text)
```

- Uses a context manager (with) for automatic file handling
- Mode 'w' opens for writing (creates or overwrites)
- encoding='utf-8' ensures proper character handling
- try/except: Handles errors (e.g., permission issues)

Lines 35-48: File Reading Section

Lines 37-44: Read Operation

*with* open(filename, 'r', encoding='utf-8') as file:

```
content = file.read()
```

- Mode 'r' opens for reading
- file.read() reads the entire file into a string
- Displays the content with formatting

Lines 45-48: Error Handling

- FileNotFoundError: Handles missing file
- Generic Exception: Handles other read errors Lines 50-51: Program Entry Point

```
if __name__ == "__main__":
```

main()

- Runs main() when executed directly
- Allows importing as a module without running

Key Concepts Demonstrated

1. Context managers: Automatic file closing
2. Error handling: Uses try/except blocks
3. File modes: 'w' (write) and 'r' (read)
4. UTF-8 encoding: Ensures international character support

This is a solid introduction to file I/O in Python with proper error handling.

### Task 3: CSV Data Analysis

#### ❖ Scenario:

**You are processing structured data from a CSV file.**

#### ❖ Task:

**Use Gemini in Colab to read a CSV file and calculate mean, min, and max.**

#### ❖ Expected Output:

➤ **Correct output**

➤ **Screenshot**

Load CSV Data

Subtask:

Load the CSV file into a pandas DataFrame.

---

Reasoning: To load the CSV file into a pandas DataFrame and display its head, I need to import the pandas library and use its read\_csv function, then call the head() method on the resulting DataFrame.

```
import pandas as pd  
df = pd.read_csv('data.csv')
```

```
df.head()
```

```
index,id,name,value
```

OUTPUT:

```
0,1,Alpha,10.0
```

```
1,2,Beta,20.0
```

```
2,3,Gamma,15.0
```

```
3,4,Delta,25.0
```

```
4,5,Epsilon,12.0
```

**\*\*Reasoning\*\*:**

The previous attempt to load `data.csv` failed because the file was not found. To resolve this, I will create a dummy `data.csv` file with some sample data using a shell command, and then retry loading it into a pandas DataFrame.

OUTPUT:

```
%%writefile data.csv
```

```
id,name,value
```

```
1,Alpha,10
```

```
2,Beta,20
```

```
3,Gamma,15
```

```
4,Delta,25
```

5,Epsilon,12

```
import pandas as pd  
df = pd.read_csv('data.csv')  
df.head()
```

Calculate Statistics

Subtask:

Calculate the mean, minimum, and maximum for all numerical columns in the DataFrame.

---

Reasoning: To calculate the mean, minimum, and maximum for all numerical columns, I will use the `.mean()`, `.min()`, and `.max()` methods on the DataFrame `df`.

```
df_mean = df.select_dtypes(include='number').mean()  
df_min = df.select_dtypes(include='number').min()  
df_max = df.select_dtypes(include='number').max()
```

```
print("Mean values:\n", df_mean)  
print("\nMinimum values:\n", df_min)  
print("\nMaximum values:\n", df_max)
```

OUTPUT:

Mean values:

value     16.4

dtype: float64

Minimum values:

value 10.0

dtype: float64

Maximum values:

value 25.0

dtype: float64

**\*\*Reasoning\*\*:**

The previous cell failed because the DataFrame `df` was not defined.

Although `data.csv` exists, the code to load it into `df` was not executed.

Therefore, I need to include the necessary steps to load `df` from `data.csv` before calculating the statistics. import pandas as pd

```
df = pd.read_csv('data.csv')
```

```
df_mean = df.select_dtypes(include='number').mean()
```

```
df_min = df.select_dtypes(include='number').min()
```

```
df_max = df.select_dtypes(include='number').max()
```

```
print("Mean values:\n", df_mean)
```

```
print("\nMinimum values:\n", df_min)
```

```
print("\nMaximum values:\n", df_max) OUTPUT:
```

Mean values:

value 16.4

dtype: float64

Minimum values:

value 10.0

dtype: float64

Maximum values:

value 25.0

dtype: float64

Final Task

Subtask:

Summarize the calculated statistics from the CSV file.

CODE EXPLANATION:

Certainly! The last code block you executed calculates and prints the mean, minimum, and maximum values for the numerical columns in your DataFrame df. Here's a breakdown:

- **import pandas as pd**: This line imports the pandas library, which is essential for working with DataFrames, and gives it the alias pd for convenience.
- **df = pd.read\_csv('data.csv')**: This reads the data from your data.csv file and loads it into a pandas DataFrame named df.
- **df\_mean = df.select\_dtypes(include='number').mean()**: This selects all columns in df that have a numerical data type, and then calculates the mean for each of those numerical columns. The result is stored in df\_mean.
- **df\_min = df.select\_dtypes(include='number').min()**: Similar to the mean, this selects numerical columns and calculates the minimum value for each, storing it in df\_min.

- **df\_max = df.select\_dtypes(include='number').max():** This selects numerical columns and calculates the maximum value for each, storing it in df\_max.
- **print(...):** These lines then print out the calculated mean, minimum, and maximum values to the console, making them easy to view.

## Task 4: Sorting Lists – Manual vs Built-in

### ❖ Scenario:

You are reviewing algorithm choices for efficiency.

### ❖ Task:

Use Gemini to generate:

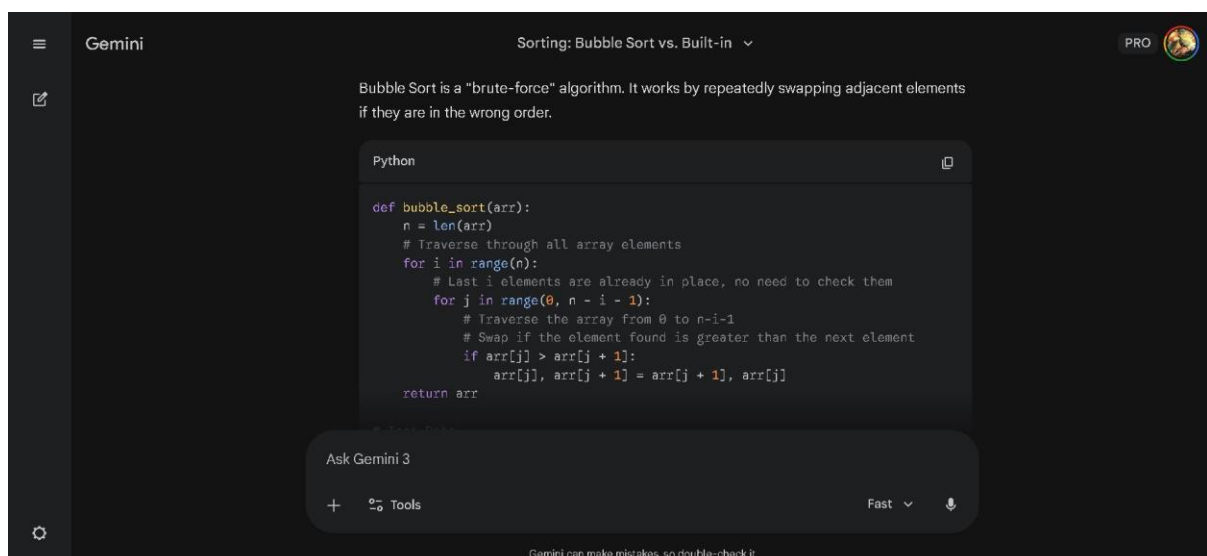
➤ Bubble sort

➤ Python's built-in sort() ➤ Compare both implementations.

### ❖ Expected Output:

➤ Two versions of code

➤ Short comparison



## 1. Manual Implementation: Bubble Sort

**Bubble Sort is a "brute-force" algorithm. It works by repeatedly swapping adjacent elements if they are in the wrong order. Python**

```
def bubble_sort(arr):  
    n = len(arr)  
    # Traverse through all array elements  
    for i in range(n):  
        # Last i elements are already in place, no need to check them  
        for j in range(0, n - i - 1):  
            # Traverse the array from 0 to n-i-1  
            # Swap if the element found is greater than the next element  
            if arr[j] > arr[j + 1]:  
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  
    return arr  
  
# Test Data  
unsorted_list = [64, 34, 25, 12, 22, 11, 90]  
result = bubble_sort(unsorted_list)  
print(result)
```

### Output:

Plaintext

[11, 12, 22, 25, 34, 64, 90]

---

## 2. Built-in Implementation: sort()

**Python uses an algorithm called Timsort. It is a hybrid approach that is significantly faster and more memory-efficient than Bubble Sort.**

**Python #**

**Test Data**



```
unsorted_list = [64, 34, 25, 12, 22, 11, 90]
```

```
# Use the built-in sort() method (modifies the list in place)
```

```
unsorted_list.sort()
```

```
print(unsorted_list)
```

**Output:**

**Plaintext**

```
[11, 12, 22, 25, 34, 64, 90]
```

---

### 3. Comparison Summary

Feature	Bubble Sort	Python sort()
Complex	$O(n^2)$ (Slow)	$O(n \log n)$ (Fast)
Effort	Manual logic required	Optimized built-in function

Feature	Bubble Sort	Python sort()
Usage	Educational / Teaching	Real-world applications

---

### 4. Code Explanation

Bubble Sort Explanation:

- Nested Loops: The outer loop (i) keeps track of how many passes we have made. The inner loop (j) compares side-by-side numbers.
- The "Bubble" Mechanism: In every pass, the largest unsorted number "bubbles up" to its correct position at the far right of the list.
- Efficiency Issue: Because it uses nested loops, if you double the number of items in a list, the time it takes to sort them quadruples.

Built-in sort() Explanation:

- Timsort Logic: Instead of just swapping, it identifies small segments of the data that are already sorted (called "runs") and merges them efficiently.
- Optimization: It is written in C, making it hundreds of times faster than a manual Python loop for large datasets.
- In-place Mutation: The .sort() method changes the original list directly, which is highly memory-efficient.

Would you like me to demonstrate how to use sorted() instead, if you want to keep your original list unchanged?

